7 Suffix arrays

This exposition was developed by Clemens Gröpl and Knut Reinert. It is based on the following sources, which are all recommended reading:

1. Simon J. Puglisi, W. F. Smyth, and Andrew Turpin, A taxonomy of suffix array construction algorithms, ACM Computing Surveys, Vol. 39, Issue 2, to appear (2007). [PST07]
2. Udi Manber, Gene Myers: Suffix arrays: A new method for online string searching, SIAM Journal on Computing 22:935-48,1993
3. Kasai, Lee, Arimura, Arikawa, Park: Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications, CPM 2001
4. Mohamed Ibrahim Abouelhoda, Stefan Kurtz, Enno Ohlebusch: Replacing suffix trees with enhanced suffix arrays. Journal of Discrete Algorithms 2 (2004) 53-86.
5. Dan Gusfield: Algorithms in strings, trees and sequences, Cambridge, pages 94ff.

7.1 Introduction

Exact string matching is a basic step used by many algorithms in computational biology: Given a pattern $P = P[1 .. m]$, and a text $S = S[1 .. n]$, we want to find all occurences of P in S.

This can readily be done with exact string matching algorithms in time $O(m+n)$. These algorithms perform some kind of *preprocessing of the pattern*. In this way it is often possible to exclude portions of the text from consideration (e. g. the Horspool algorithm can shift the search window by m positions if a verification fails). But as long as $m = O(1)$, the running time for this class of algorithms cannot be $o(n)$.

In order to achieve a truly sublinear search time, we have to *preprocess the text*. Preprocessing the text is useful in scenarios where the text is relatively constant over time (e. g. a genome), and we will search for many different patterns.

Even if the text is very long, we do not need to scan it completely for every query. The running time can be as low as $O(m + p)$, where p is the number of occurrences. Here we will see algorithms to achieve a search time of $O(m + p + \log n)$. In practice, the extra $\log n$ factor is counterbalanced by a good caching behavior.

In this lecture we introduce one such preprocessing, namely the construction of a *suffix array*.

Suffix arrays are closely related to suffix trees. A good reference for suffix trees is the book of Gusfield. In 1990, Manber and Myers introduced suffix arrays as a space efficient alternative to suffix trees.

Both suffix trees and suffix arrays require $O(n)$ space, but whereas a recent, tuned suffix tree implementation requires 13-15 Bytes per character (Kurtz, 1999), for suffix arrays, as few as $5 + o(1)$ bytes are sufficient (with some tricks).

Definition 1. Given a text S of length n, the *suffix array* for S, often denoted *suftab*, is an array of integers of range 1 to n specifying the lexicographic ordering of the suffixes of the string S.

It will be convenient to assume that $S[n] = \$$, where \$ is smaller than any other letter.

That is, suftab$[j] = i$ if and only if $S[i .. n]$ is the j-th suffix of S in ascending lexicographical order. We will write $S_i := S[i .. n]$.

We will assume that n fits into 4 bytes of memory. (That is, $n < 2^{32} = 4\,294\,967\,296$.) Then the basic form of a suffix array needs only $4n$ bytes.

The suffix array can be computed by sorting the suffixes, as illustrated in the following example.

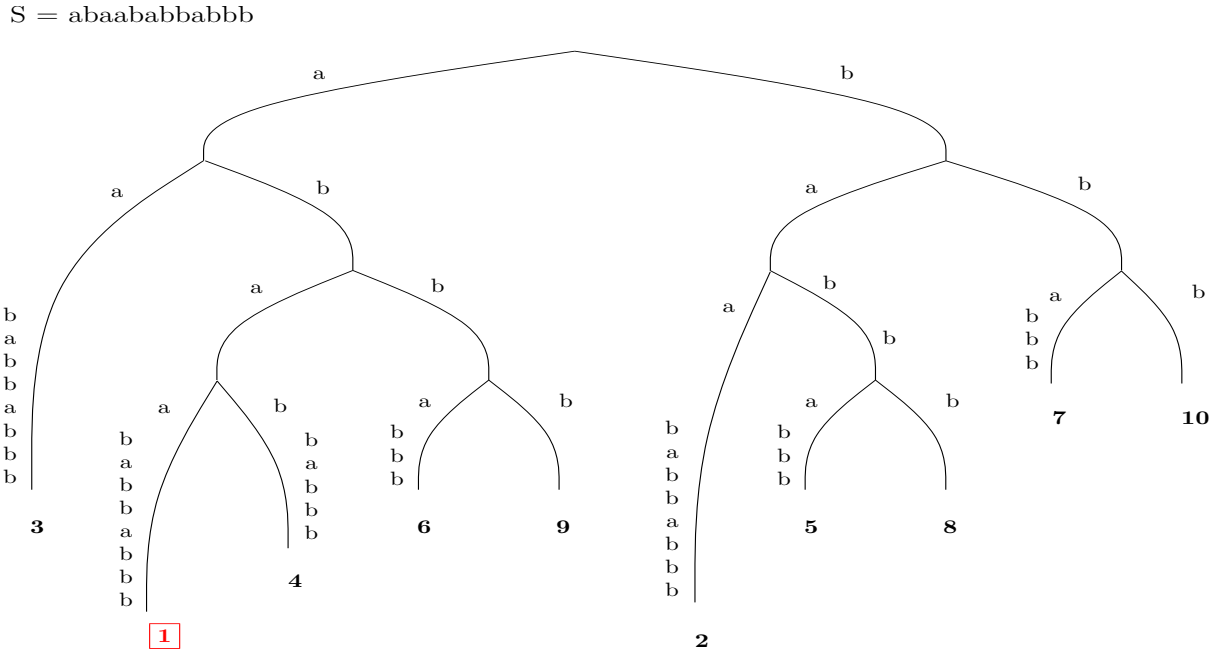
7.2 Example

The text is S = *abaababbabbb*\$, $n = 13$. The suffix array is:

Suffixes		Ordered suffixes		
i	S_i	i	suftab[i]	$S_{\text{suftab}[i]}$
1	abaababbabbb\$	1	13	\$
2	baababbabbb\$	2	3	aababbabbb\$
3	aababbabbb\$	3	1	abaababbabbb\$
4	ababbabbb\$	4	4	ababbabbb\$
5	babbabbb\$	5	6	abbabbb\$
6	abbabbb\$	6	9	abbb\$
7	bbabbb\$	7	12	b\$
8	babbb\$	8	2	baababbabbb\$
9	abbb\$	9	5	babbabbb\$
10	bbb\$	10	8	babbb\$
11	bb\$	11	11	bb\$
12	b\$	12	7	bbabbb\$
13	\$	13	10	bbb\$

It is tempting to confuse *suftab*[i] with $S_{\text{suftab}[i]}$ since there is a one-to-one correspondence, but of course the two are completely different concepts.

Compare this to the suffix tree, which is obtained by merging common prefixes of the suffixes S_i in a trie. (Note: The string in the figure has no trailing \$. Some suffixes are not numbered; their paths lead to internal nodes.)



7.3 Why another algorithm?

The suffix array can be constructed in (essentially) $4n$ space by sorting the suffix indices using any sorting algorithm. (*Exercise*: How much would a simple quicksort cost?) But such an approach fails to take advantage of the fact that we are sorting a collection of related suffixes. We cannot get an $O(n)$ time algorithm in this way.

Alternatively, we could first build a suffix tree in linear time, then transform the suffix tree into a suffix array in linear time (*exercise*: work out the details), and finally discard the suffix tree. Of course, sufficient memory has to be available to construct the suffix tree. Thus this approach fails for large texts.

Over the last 15 years or so, there have been hundreds of research articles published on the construction

and application of suffix trees and suffix arrays. A recent survey on suffix array construction algorithms is [PST07]. In the introduction, Puglisi, Smyth, and Turpin write:

> *It has been shown that*
>
> - *practical space-efficient suffix array construction algorithms (SACAs) exist that require worst-case time linear in string length;*
> - *SACAs exist that are even faster in practice, though with supralinear worstcase construction time requirements;*
> - *any problem whose solution can be computed using suffix trees is solvable with the same asymptotic complexity using suffix arrays.*
>
> *Thus suffix arrays have become the data structure of choice for many, if not all, of the string processing problems to which suffix tree methodology is applicable.*

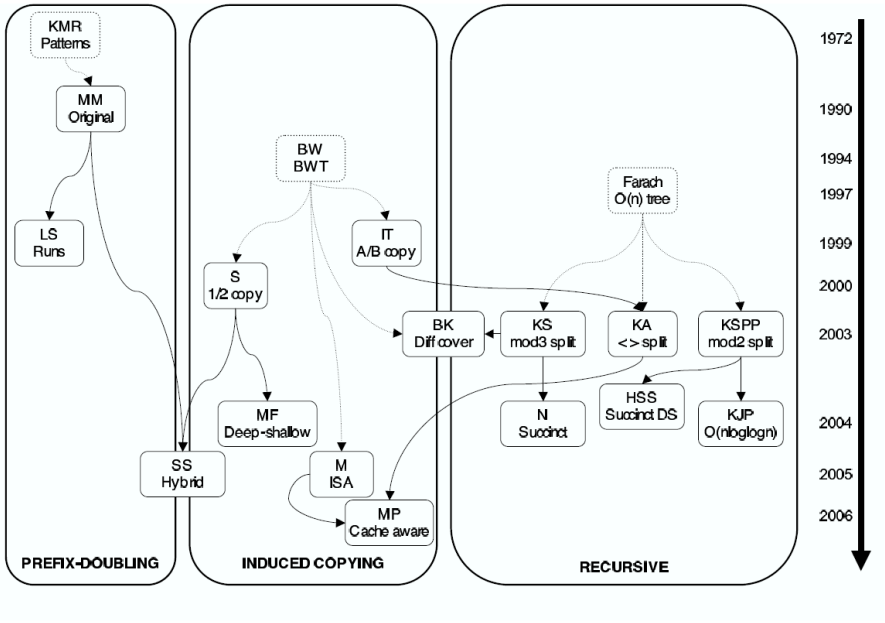


Fig. 2. Taxonomy of suffix array construction algorithms [PST07]

In [PST07] running times for 17 SACAs are listed. Today the original construction proposed by Manber and Myers is about 30 times slower than the fastest SACA known so far. The race is not finished yet, new algorithms and implementations are being developed and it is hard to predict where this will eventually lead to. Therefore we will not discuss a SACA in full detail in this lecture but only mention a few basic ideas.

One such idea is *prefix doubling*. It is the fundament of the original MM algorithm (1990). A modified version by Larsson and Sadakane (1999) is 'only' a factor 3 slower than the currently best one.

7.4 Prefix doubling

In order to construct the suffix array we have to cleverly sort the n suffixes $S_1, \ldots, S_n$.

A prefix-doubling algorithm will not sort the suffixes completely in a single stage. Instead, it proceeds in $\lceil \log_2(n+1) \rceil$ stages.

In the first stage the suffixes are arranged into groups or *buckets* according to their first symbol. Thus they are ordered with repect to their prefixes of length 1.

We say that the suffixes are in $\leq_h$*-order* if they are ordered lexicographically according to the first h letters ($=_h$ and $<_h$ are defined accordingly).

Inductively, the algorithm partitions the buckets of the preceeding stage ($\leq_h$) further by sorting according to *twice* the number of symbols ($\leq_{2h}$). We will number the stages $1, 2, 4, 8, \ldots$ to indicate the number of affected symbols. After the h-th stage, the suffixes are sorted according to $\leq_h$ order, and all suffixes in a bucket have a common prefix of length h.

We are done when $h \geq n$. Each stage takes $O(n)$ time. Thus the total running time is $O(n \log n)$.

The key observation is:

- In order to refine the ordering of an h-bucket to a $\leq_{2h}$-order, it suffices to look at the h positions following the (common) prefix of length h;
- These positions are the prefixes of other suffixes and have been $\leq_h$-sorted already.

This technique has become known as *prefix doubling*.

Let us summarize this idea:

Observation 2 (Karp, Miller, Rosenberg (1972))**.**
Let S_i and S_j be two suffixes belonging to the same bucket after the h-th step, that is $S_i =_h S_j$. We need to compare the next h symbols. But the next h symbols of S_i (respectively, S_j) are exactly the first h symbols of S_{i+h} (respectively, S_{j+h}). By assumption we already know the relative order of S_{i+h} and S_{j+h} according to $\leq_h$.

For this approach to work it is necessary that we can access the $\leq_h$*-rank* of a suffix (i. e., its position according to the $\leq_h$-order). Therefore the inverse of the current *suftab* table is stored in another table *sufinv*.

These two tables (suftab, sufinv) together amount to the $8n$ bytes required by the Manber-Myers algorithm.

7.5 Searching

After constructing our suffix array we have the table *suftab* which gives us in sorted order the suffixes of S. Suppose now we want to find all instances of a string $P = p_1, \ldots, p_m$ of length $m < n$ in S.

Let

$$L_P = \min\{k : P \leq_m S_{suftab[k]} \text{ or } k = n+1\}$$

and

$$R_P = \max\{k : S_{suftab[k]} \leq_m P \text{ or } k = 0\}.$$

Since *suftab* is in $\leq_m$ order, it follows that P matches a suffix S_i if and only if $i =$ *suftab*$[k]$ for some $k \in [L_P, R_P]$. Hence a simple binary search can find L_P and R_P. Each comparison in the search needs $O(m)$ character comparisons, and hence we can find all instances in the string in time $O(m \log n)$.

This is the simple code piece to search fo L_P.

```
1  if P ≤_m S_suftab[1]
2    then L_P = 1;
3    else if P >_m S_suftab[n]
4           then L_P = n + 1;
5           else
6                (L, R) = (1, n);
7                while R − L > 1 do
8                      M = ⌈(L + R)/2⌉;
9                      if P ≤_m S_suftab[M]
10                       then R = M;
11                       else L = M;
12                     fi
13               od
14               L_P = R;
15         fi
16 fi
```

For example if we search for $P = aca$ in the text $S = acaaacatat\$$ then $L_P = 3$ and $R_P = 4$. We find the value L_P and R_P respectively, by setting (L, R) to $(1, n)$ and changing the borders of this interval based on the comparison with the suffix at position $\lceil (L+R)/2 \rceil$ e.g. we find L_P with the sequence: $(1, 11) \Rightarrow (1, 6) \Rightarrow (1, 4) \Rightarrow (1, 3) \Rightarrow (2, 3)$. Hence $L_P = 3$.

1	aaacatat$
2	aacatat$
3	acaaacatat$
4	acatat$
5	atat$
6	at$
7	caaacatat$
8	catat$
9	tat$
10	t$
11	$

The binary searches each need $O(\log n)$ steps. In each step we need to compare m characters of the text and the pattern in the $\geq_m$ operations. This leads to a running time of $O(m \log n)$.

Can we do better?

While the binary search continues, let L and R denote the left and right boundaries of the current search interval. At the start, L equals 1 and R equals n. Then in each iteration of the binary search a query is made at location $M = \lceil (R + L)/2 \rceil$ of $suftab$.

We keep track of the longest prefixes of $S_{suftab(L)}$ and $S_{suftab(R)}$ that match a prefix of P. Let l and r denote the prefix lengths respectively and let $mlr = \min(l, r)$.

Then we can use the value mlr to accelerate the lexicographical comparison of P and the suffix $S_{suftab[M]}$. Since $suftab$ is ordered, it is clear that all suffixes between L and R share the same prefix. Hence we can start the first comparison at position $mlr + 1$.

In practice this trick already brings the running time to $O(m + \log n)$, however one can construct an example that still needs time $O(m \cdot \log n)$ (exercise).

We call an examination of a character of P *redundant* if that character has been examined before. The goal is to limit the number of redundant character comparisons to $O(1)$ per iteration of the binary search.

The use of mlr alone does not suffice: In the case that $l \neq r$, all characters in P from $mlr + 1$ to $\max(l, r)$ will have already been examined. Thus all comparisons to these characters are redundant. We need a way to begin the comparisons at the *maximum* of l and r.

To do this we introduce the following definition.

Definition 3. $lcp(i, j)$ is the length of the longest common prefix of the suffixes specified in positions i and j of $suftab$.

For example for $S = aabaacatat$ the $lcp(1, 2)$ is the length of the longest common prefix of *aabaacata* and *aacata* which is 2.

With the help of the lcp information, we can achieve our goal of one redundant character comparison per iteration of the search. For now assume that we know $lcp(i, j), \forall i, j$.

How do we use the lcp information? In the case of $l = r$ we compare P to $suftab[M]$ as before starting from position $mlr + 1$, since in this case the minimum of l and r is also the maximum of the two and no redundant character comparisons are made.

If $l \neq r$, there are three cases. We assume w.l.o.g. $l > r$.

Case 1: $lcp(L, M) > l$.
Then the common prefix of the suffixes $S_{suftab[L]}$ and $S_{suftab[M]}$ is longer than the common prefix of P and $S_{suftab[L]}$.

Therefore, P agrees with the suffix $S_{suftab[M]}$ up through character l. Or to put it differently, characters $l + 1$ of $S_{suftab[L]}$ and $S_{suftab[M]}$ are identical and lexically less than character $l + 1$ of P.

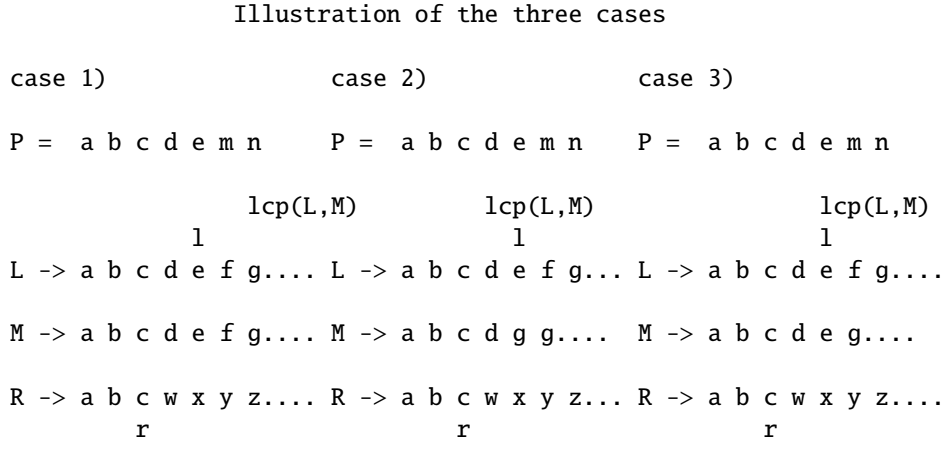
Hence any possible starting position must start to the right of M in $suftab$. So in this case *no* examination of P is needed. L is set to M and l and r remain unchanged.

Case 2: $lcp(L, M) < l$.
Then the common prefix of suffix $suftab[L]$ and $suftab[M]$ is smaller than the common prefix of $suftab[L]$ and P.

Therefore P agrees with $suftab[M]$ up through character $lcp(L,M)$. The $lcp(L,M)+1$ characters of P and and $suftab[L]$ are identical and lexically less than the character $lcp(L,M)+1$ of $suftab[M]$.

Hence any possible starting position must start left of M in $suftab$. So in this case again *no* examination of P is needed. R is set to M, r is changed to $lcp(L,M)$, and l remains unchanged.

Case 3: $lcp(L,M) = l$.
Then P agrees with $suftab[M]$ up to character l. The algorithm then lexically compares P to $suftab[M]$ starting from position $l+1$. In the usual manner the outcome of the compare determines which of L and R change along with the corresponding change of l and r.



Then the following theorem holds:

Theorem 4. *Using the lcp values, the search algorithm does at most $O(m + \log n)$ comparisons and runs in that time.*

Proof: Exercise. Use the fact that neither l nor r decrease in the binary search, and find a bound for the number of redundant comparisons per iteration of the binary search.

7.6 Computing the *lcp* values

We now know how to search fast in a suffix array under the assumption, that we know the *lcp* values for all pairs i, j.

But how do we compute the *lcp* values? And which ones? Computing them all would require too much time and, worse, quadratic space!

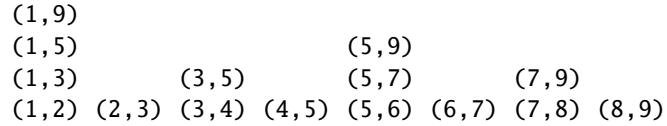
We will now first dicuss, which *lcp* values we really need, and then how to compute them. For the computation give in more detail a newer, simple $O(n)$ algorithm to compute the *lcp* values given the suffix array $suftab$.

In the appendix we also sketch Myers' proposal for computing the *lcp* values during the construction of the suffix array.

We first observe that indeed we only need the *lcp* values of L and R that we encounter in the binary search for L_P and R_P. However, the set of pairs (i, j) which can be considered is contained in a binary search tree which does not depend on P, and has linear size.

Observation 5. Only $O(n)$ many *lcp* values are needed for the *lcp* based search in a suffix array.

Example: $n = 9$



We get those values in a two step procedure:

1. Compute the *lcp* values for pairs of suffixes *adjacent* in *suftab* using an array *height* of size n.
2. For the fixed binary search tree used in the search for L_P and R_P compute the *lcp* values for its internal nodes using the array *height*. (exercise *)

(*) The value at an internal node is the minimum of its successors (why?)

Hence the essential thing to do is to compute the array *height*, i.e. the *lcp* values of adjacent suffixes in *suftab*.

7.7 The Kasai et al. algorithm

An elegant, short algorithm for computing the *height* array in linear time is due to Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park (presented at CPM 2001).

The array *height* is defined by

$$height(k) = lcp(S_{suftab[k-1]}, S_{suftab[k]}).$$

That is, it contains the *lcp* values of all adjacent suffixes in the suffix array *suftab*.

We can compute the *lcp* values contained in the binary search tree in linear time and space, once we have *height* values.

The Kasai et al. algorithm uses the inverse of the suffix array, that is, the array *sufinv* with the defining property

$$sufinv[suftab[i]] = i.$$

Clearly, *sufinv* can be computed in one linear scan over *suftab*, if it is not available yet.

It is important to keep the "semantics" of *suftab* and *sufinv* in mind. Perhaps the following diagram is useful:

$$\begin{array}{ccc} sufinv[i] = j & \Leftrightarrow & \text{Suffix } S_i \text{ has rank } j \text{ in lexicographic order} \\ \Updownarrow & & \Updownarrow \\ suftab[j] = i & \Leftrightarrow & j\text{-th lowest Suffix in lexicographic order is } S_i \end{array}$$

The algorithm computes the *height* values of the suffixes S_i in order of decreasing length. Thus the main loop runs over $i = 1, \ldots, n$.

Let $p := sufinv(i)$. The *height* value for S_i depends on S_i and its predecessor in *suftab*; we have

$$height(p) = lcp(S_{suftab(p-1)}, S_{suftab(p)}) = lcp(S_k, S_i),$$

with $k := suftab(p-1)$.

i	$suftab[i]$
$p-1$	k
p	i

The algorithm keeps track of the last *height* value computed, h. Initially, we have $h = 0$. Then $height(p)$ is computed in the straight-forward way:

```
1 while S[i + h] = S[k + h] do
2       h++;
3 od
4 height[sufinv[i]] = h;
```

From now on, we assume that the last h has been computed correctly.

Now the algorithm proceeds to S_{i+1}.

But in fact $height(sufinv(i))$ and $height(sufinv(i+1))$ are closely related. Namely, if $h = height(sufinv(i)) > 0$, then

$$lcp(S_i, S_{suftab[sufinv[i]-1]}) = h > 0$$

and hence,

$$lcp(S_{i+1}, S_{suftab[sufinv[i]-1]+1}) = h - 1\,.$$

Moreover,

$$S_i \geq_{\text{lex.}} S_{suftab[sufinv[i]-1]}$$

implies

$$S_{i+1} \geq_{\text{lex.}} S_{suftab[sufinv[i]-1]+1}\,,$$

because the first letters were the same.

Now, how does this relate to $height(sufinv(i+1))$?

Let $p' := sufinv(i+1)$. By the preceeding observation, we have found a position $q' < p'$ such that

$$lcp(S_{suftab[q']}, S_{suftab[p']}) \geq h - 1\,,$$

namely, $q' := sufinv[suftab[sufinv[i]-1]+1]$. But we cannot assert that q' is the *immediate* predecessor of p'.

i	$suftab[i]$	
$p-1$	k	$k = suftab[sufinv[i]-1]$
p	i	$p = sufinv[i]$
$\vdots$		
$\vdots$		
q'	$k+1$	$q' = sufinv[suftab[sufinv[i]-1]+1]$
$\vdots$		(maybe $q' < p'-1$)
p'	$i+1$	$p' = sufinv[i+1]$

Yet the following observation helps. We have

$$\begin{aligned} h-1 &\leq lcp(S_{suftab[q']}, S_{suftab[p']}) \\ &= \min_{k' \in [q', p'-1]} lcp(S_{suftab[k]}, S_{suftab[k+1]}) \\ &\leq lcp(S_{suftab[p'-1]}, S_{suftab[p']}) \\ &= height(p')\,. \end{aligned}$$

In other words, the next $height$ value to be computed (belonging to S_{i+1}) is at most *one* less than the preceeding one (belonging to S_i).

7.8 The algorithm

The following algorithm computes the array $height$ following the above discussion in time $O(n)$:

```
1  GetHeight(S, suftab)
2  for i = 1 to n do
3      sufinv[suftab[i]] = i;
4  od
5  h = 0;
6  for i = 1 to n do
7      if sufinv[i] > 1
8        then
9              k = suftab[sufinv[i] − 1];
10             while S[i + h] = S[k + h] do
11                   h++;
12             od
13             height[sufinv[i]] = h;
14             if h > 0 then h = h − 1; fi
15     fi
16 od
```

The above algorithm uses only linear time. In the loop in line 6 we iterate from 1 to n. In the loop is a while loop in line 10 that increases the height (i.e. the *lcp* value of adjacent suffixes). Since the height is maximally n and since in line 14 we decrease h by at most 1 per iteration of the main loop, it follows that the while loop can increase h at most $2n$ times in total.

7.9 Example

The example was prepared using Stefan Kurtz's programs `mkvtree` and `vstree2tex`, see `www.vmatch.de`. (Sorry for index shifts!)

i	$suftab[i]$	$height[i]$	$S_{suftab[i]}$
0	2		aaacatat$
1	3		aacatat$
2	0	1	acaaacatat$
3	4		acatat$
4	6		atat$
5	8		at$
6	1		caaacatat$
7	5		catat$
8	7		tat$
9	9		t$
10	10		$

i	$sufinv[i]$	S_i
0	$*$ 2	acaaacatat$
1	6	caaacatat$
2	0	aaacatat$
3	1	aacatat$
4	3	acatat$
5	7	catat$
6	4	atat$
7	8	tat$
8	5	at$
9	9	t$
10	10	$

$i = 0$: $sufinv[i] = 2$, $k = suftab[\,sufinv[i] - 1] = suftab[1] = 3$. We compare S_0 and S_3 and get $height[2] = 1$.

i	$suftab[i]$	$height[i]$	$S_{suftab[i]}$
0	2		aaacatat$
1	3		aacatat$
2	0	1	acaaacatat$
3	4		acatat$
4	6		atat$
5	8		at$
6	1	0	caaacatat$
7	5		catat$
8	7		tat$
9	9		t$
10	10		$

i	$sufinv[i]$	S_i
0	2	acaaacatat$
1	$*$ 6	caaacatat$
2	0	aaacatat$
3	1	aacatat$
4	3	acatat$
5	7	catat$
6	4	atat$
7	8	tat$
8	5	at$
9	9	t$
10	10	$

$i = 1$: $sufinv[i] = 6$, $k = suftab[\,sufinv[i] - 1] = suftab[5] = 8$. We compare S_1 and S_8 and get $height[6] = 0$.

i	$suftab[i]$	$height[i]$	$S_{suftab[i]}$
0	2	-/-	aaacatat$
1	3		aacatat$
2	0	1	acaaacatat$
3	4		acatat$
4	6		atat$
5	8		at$
6	1	0	caaacatat$
7	5		catat$
8	7		tat$
9	9		t$
10	10		$

i	$sufinv[i]$	S_i
0	2	acaaacatat$
1	6	caaacatat$
2	$*$ 0	aaacatat$
3	1	aacatat$
4	3	acatat$
5	7	catat$
6	4	atat$
7	8	tat$
8	5	at$
9	9	t$
10	10	$

$i = 2$: $sufinv[i] = 0$. There is no *height* value in the first row.

i	$suftab[i]$	$height[i]$	$S_{suftab[i]}$
0	2		aaacatat$
1	3	2	aacatat$
2	0	1	acaaacatat$
3	4		acatat$
4	6		atat$
5	8		at$
6	1	0	caaacatat$
7	5		catat$
8	7		tat$
9	9		t$
10	10		$

i	$sufinv[i]$	S_i
0	2	acaaacatat$
1	6	caaacatat$
2	0	aaacatat$
3	$*1$	aacatat$
4	3	acatat$
5	7	catat$
6	4	atat$
7	8	tat$
8	5	at$
9	9	t$
10	10	$

$i = 3$: $sufinv[i] = 1$, $k = suftab[\,sufinv[i] - 1] = suftab[0] = 2$. We compare S_3 and S_2 and get $height[1] = 2$.

i	$suftab[i]$	$height[i]$	$S_{suftab[i]}$
0	2		aaacatat$
1	3	2	aacatat$
2	0	1	acaaacatat$
3	4	3	acatat$
4	6		atat$
5	8		at$
6	1	0	caaacatat$
7	5		catat$
8	7		tat$
9	9		t$
10	10		$

i	$sufinv[i]$	S_i
0	2	acaaacatat$
1	6	caaacatat$
2	0	aaacatat$
3	1	aacatat$
4	$*3$	acatat$
5	7	catat$
6	4	atat$
7	8	tat$
8	5	at$
9	9	t$
10	10	$

$i = 4$: $sufinv[i] = 3$, $k = suftab[\,sufinv[i] - 1] = suftab[2] = 0$. We compare S_4 and S_0. We start at $h = lcp(S_3, S_2) - 1 = 1$. Observe that $S_4 > S_0 > S_3$ in lex. order. We get $height[3] = 3$.

$i = 5$: $sufinv[i] = 7$. We can skip the first $height[3] - 1 = 3 - 1 = 2$ letters from comparison. [...]

The final result is:

i	$suftab[i]$	$height[i]$	$S_{suftab[i]}$
0	2		aaacatat$
1	3	2	aacatat$
2	0	1	acaaacatat$
3	4	3	acatat$
4	6	1	atat$
5	8	2	at$
6	1	0	caaacatat$
7	5	2	catat$
8	7	0	tat$
9	9	1	t$
10	10	0	$

i	$sufinv[i]$	S_i
0	2	acaaacatat$
1	6	caaacatat$
2	0	aaacatat$
3	1	aacatat$
4	3	acatat$
5	7	catat$
6	4	atat$
7	8	tat$
8	5	at$
9	9	t$
10	10	$

Our overall strategy for constructing and searching a suffix array could then be as follows:

- Construct the suffix array for S in time $O(n \log n)$. (Linear time constructions are possible)
- Compute the *height* array (for adjacent positions) in linear time.
- Precompute the search tree for the binary search and annotate its internal nodes with *lcp* values in time $O(n)$. (exercise)
- Support $O(\log n + m)$ queries by adapting the searches for L_P and R_P.

7.10 Summary

- Suffix arrays are a space efficient alternativ to suffix trees.
- They can be built in time $O(n \log n)$.
- Simple searches can be conducted in time $O(m \log n)$. However the simple *mlr* heuristic perofrms already well in practice (time $O(m + \log n)$.
- The *lcp* values can be computed in linear time, given a suffix array.
- Using the *lcp* values the search in suffix arrays can be speeded up to $O(m + \log n)$.

7.11 Appendix: Manber-Myers algorithm

The Manber-Myers algorithm stores the result in the table *suftab* and, in addition, uses in another array *Bh* of boolean values to demarcate the partitioning of the suffix array into buckets. Each bucket initially holds the suffixes with the same first symbol.

The algorithm uses some auxiliary boolean tables. These are stored as higher-order bits in the other tables and thus do not require additional memory allocation. However, the range of feasible n is reduced to 2^{31} if this implementation technique is used.

The algorithm needs (essentially) $8n$ bytes and runs in $O(n \log n)$ time.

```
initialize SA_1, ISA_1
while some h-group not a singleton
  for j ← 1 to n do
    i ← SA_h[j] − h
    if i > 0 then
      q ← head[h-group[i]]
      SA_2h[q] ← i
      head[h-group[i]] ← q+1
  compute ISA_2h  —  update 2h-groups
  h ← 2h
```

Fig. 3. Algorithm MM [PST07]

Attention: There is an index shift in the following presentation, *suftab* starts at position 0.

If we look at the example *acbaacatat*\$, we have the following after stage 1 (extra space separates the *h*-buckets):

i	*Bh* [i]	*suftab* [i]
0	1	0=acbaacatat$
1	0	3=aacatat$
2	0	4=acatat$
3	0	6=atat$
4	0	8=at$
5	1	2=baacatat$
6	1	1=cbaacatat$
7	0	5=catat$
8	1	7=tat$
9	0	9=t$
10	1	10=$

The idea is now the following: Let S_i be the first suffix in the first bucket (i.e. $suftab[0] = i$), and consider S_{i-h}.

Since S_i starts with the smallest h-symbol string, S_{i-h} should be the first in its $2h$ bucket. Hence we move S_{i-h} to the beginning of its bucket and mark this fact. Remember this:

> *The algorithm scans the suffixes S_i as they appear in $\leq_h$-order. For each S_i, it moves S_{i-h} to the next available place in* its *h-bucket.*

Altogether, we maintain three integer arrays *suftab*, *sufinv* and *count*, and two boolean arrays *Bh* and *B2h*, all with $n + 1$ elements.

At the start of stage h, *suftab*$[i]$ contains the start position of the i-th smallest suffix (according to the first h symbols).

sufinv$[i]$ is the inverse of suftab, i. e.

$$suftab[\,sufinv[\,i\,]\,] = i\,.$$

Bh$[i]$ is 1 iff *suftab*$[i]$ contains the leftmost suffix of an h-bucket.

(The actual implementation of *count*, *Bh*, and *B2h* uses bits and currently unused entries from *suftab* and *sufinv*.)

If we look at the example *acbaacatat*\$, we have the following:

i	*Bh* [i]	*sufinv* [i]	*suftab* [i]
0	1	0	0=acbaacatat$
1	0	6	3=aacatat$
2	0	5	4=acatat$
3	0	1	6=atat$
4	0	2	8=at$
5	1	7	2=baacatat$
6	1	3	1=cbaacatat$
7	0	8	5=catat$
8	1	4	7=tat$
9	0	9	9=t$
10	1	10	10=$

In stage $2h$ we reset *sufinv*$[i]$ to point to the leftmost cell of the h-bucket containing the i-th suffix, rather than to the suffix's precise place in the bucket. In our example we get:

i	*Bh* [i]	*sufinv* [i]	*suftab* [i]
0	1	0	0=acbaacatat$
1	0	6	3=aacatat$
2	0	5	4=acatat$
3	0	0	6=atat$
4	0	0	8=at$
5	1	6	2=baacatat$
6	1	0	1=cbaacatat$
7	0	8	5=catat$
8	1	0	7=tat$
9	0	8	9=t$
10	1	10	10=$

In each doubling step, *suftab* is scanned in increasing order, one bucket at a time. Let l and r mark the left and right boundary of the h-bucket currently being scanned. For every $l \leq i \leq r$, we do the following:

1. Let $T_i := suftab[i] - h$. (If T_i is negative we do nothing.)
2. Increment $count[sufinv[T_i]]$.
3. Set $sufinv[T_i] = sufinv[T_i] + count[sufinv[T_i]] - 1$.
4. Mark this by setting $B2h[sufinv[T_i]]$ to 1.

Now $sufinv[i]$ is correct with respect to $\leq_{2h}$. The old $\leq_h$-ordering is still available in *suftab*. The *suftab* is updated at the end of the $2h$ stage. In the following example, we show the future positions of the suffixes a field *new_st* (not used by the algorithm).

We indicate the current position with a "*". The auxiliary array *count* is initialized to 0 for all i. After the initialization:

i	*Bh* [i]	*B2h* [i]	*count* [i]	*sufinv* [i]	*new_st*[i]	*suftab* [i]
* 0	1	0	0	0		0=acbaacatat$
1	0	0	0	6		3=aacatat$
2	0	0	0	5		4=acatat$
3	0	0	0	0		6=atat$
4	0	0	0	0		8=at$
5	1	0	0	6		2=baacatat$
6	1	0	0	0		1=cbaacatat$
7	0	0	0	8		5=catat$
8	1	0	0	0		7=tat$
9	0	0	0	8		9=t$
10	1	0	0	10		10=$

Nothing happens since $0 - 1 < 0$.

i	*Bh* [i]	*B2h* [i]	*count* [i]	*sufinv* [i]	*new_st*[i]	*suftab* [i]
0	1	0	0	0		0=acbaacatat$
* 1	0	0	0	6		3=aacatat$
2	0	0	0	**5**		4=acatat$
3	0	0	0	0		6=atat$
4	0	0	0	0		8=at$
5	1	1	1	6	5	2=baacatat$
6	1	0	0	0		1=cbaacatat$
7	0	0	0	8		5=catat$
8	1	0	0	0		7=tat$
9	0	0	0	8		9=t$
10	1	0	0	10		10=$

S_2 is moved to the front of its bucket, i. e. it stays were it is: $T_1 = 2$ and $sufinv[2] = 5$, hence increment $count[5]$, set $sufinv[2] = 5 + count[5] - 1 = 5$, and $B2h[5] = 1$.

i	*Bh* [i]	*B2h* [i]	*count* [i]	*sufinv* [i]	*new_st*[i]	*suftab* [i]
0	1	1	1	0		0=acbaacatat$
1	0	0	0	6	0	3=aacatat$
* 2	0	0	0	**5**		4=acatat$
3	0	0	0	**0**		6=atat$
4	0	0	0	0		8=at$
5	1	1	1	6	5	2=baacatat$
6	1	0	0	0		1=cbaacatat$
7	0	0	0	8		5=catat$
8	1	0	0	0		7=tat$
9	0	0	0	8		9=t$
10	1	0	0	10		10=$

S_3 is moved to the front of its bucket, i. e. to position 0: $T_2 = 3$ and $sufinv[3] = 0$, hence increment $count[0]$, set $sufinv[3] = 0 + count[0] - 1 = 0$, and $B2h[0] = 1$.

i	*Bh* [i]	*B2h* [i]	*count* [i]	*sufinv* [i]	*new_st*[i]	*suftab* [i]
0	1	1	1	0		0=acbaacatat$
1	0	0	0	6	0	3=aacatat$
2	0	0	0	**5**		4=acatat$
* 3	0	0	0	**0**		6=atat$
4	0	0	0	0		8=at$
5	1	1	1	**6**	5	2=baacatat$
6	1	1	1	0		1=cbaacatat$
7	0	0	0	8	6	5=catat$
8	1	0	0	0		7=tat$
9	0	0	0	8		9=t$
10	1	0	0	10		10=$

S_5 is moved to the front of its bucket, i. e. to position 6: $T_3 = 5$ and $sufinv[5] = 6$, hence increment $count[6]$, set $sufinv[5] = 6 + count[6] - 1 = 6$, and $B2h[6] = 1$.

i	*Bh* [i]	*B2h* [i]	*count* [i]	*sufinv* [i]	*new_st*[i]	*suftab* [i]
0	1	1	1	0		0=acbaacatat$
1	0	0	0	6	0	3=aacatat$
2	0	0	0	**5**		4=acatat$
3	0	0	0	**0**		6=atat$
* 4	0	0	0	0		8=at$
5	1	1	1	**6**	5	2=baacatat$
6	1	1	1	0		1=cbaacatat$
7	0	0	0	**8**	6	5=catat$
8	1	1	1	0	8	7=tat$
9	0	0	0	8		9=t$
10	1	0	0	10		10=$

S_7 is moved to the front of its bucket, i. e. to position 8: $T_4 = 7$ and $sufinv[7] = 8$, hence increment $count[8]$, set $sufinv[7] = 8 + count[8] - 1 = 8$, and $B2h[8] = 1$.

Now we have scanned the first bucket. Next we scan the bucket again, find all moved suffixes in all buckets and update the $B2h$ bitvector so that it points only to the leftmost positions of the $2h$-buckets.

To do that $B2h$ is set to false in the interval $[\,sufinv[a] + 1, b - 1]$ where a is every position marked in $B2h$ and

$$b = \min\left\{j : j > sufinv[a] \text{ and}(\,Bh[j] \text{ or not } B2h[j])\right\}.$$

It is clear that the left border preserves the leftmost bit set. The definition of the right border prevents us from resetting a border of an adjacent $2h$ bucket, but ensures the cancelling of all unwanted bits.

In our example nothing happens, since all moved suffixes were put at the beginning of a new bucket. This scan updates the $sufinv$ and $B2h$ tables and makes them consistent with the $\leq_{2h}$ order. At the end of each stage after all buckets are scanned, we update the $suftab$ array using the $sufinv$ array:

$$\text{For all } i\text{:} \qquad suftab[\,sufinv[\,i\,]\,] := i\,.$$

The next step shows that indeed the order of S_1 and S_5 is changed. S_5 was investigated during the scan of the first bucket and put to the beginning of its $\leq_{2h}$-bucket. Also, the $B2h$ vector changes now in the second scanning step.

i	*Bh* [i]	*B2h* [i]	*count* [i]	*sufinv* [i]	*new_st*[i]	*suftab* [i]
0	1	1	1	0		0=acbaacatat$
1	0	0	0	**7**	0	3=aacatat$
2	0	0	0	**5**		4=acatat$
3	0	0	0	**0**		6=atat$
4	0	0	0	0		8=at$
* 5	1	1	1	**6**	5	2=baacatat$
6	1	1	2	0	7	1=cbaacatat$
7	0	1	0	**8**	6	5=catat$
8	1	1	1	0	8	7=tat$
9	0	0	0	8		9=t$
10	1	0	0	10		10=$

Now S_1 is put at the second position of its bucket: $T_5 = 1$ and *sufinv*[1] = 6, hence increment *count*[6], set *sufinv*[1] = 6 + *count*[6] − 1 = 7, and *B2h*[7] = 1.

i	*Bh* [i]	*B2h* [i]	*count* [i]	*sufinv* [i]	*new_st*[i]	*suftab* [i]
0	1	1	2	**1**	1	0=acbaacatat$
1	0	1	0	**7**	0	3=aacatat$
2	0	0	0	**5**		4=acatat$
3	0	0	0	**0**		6=atat$
4	0	0	0	0		8=at$
5	1	1	1	**6**	5	2=baacatat$
* 6	1	1	2	0	7	1=cbaacatat$
7	0	1	0	**8**	6	5=catat$
8	1	1	1	0	8	7=tat$
9	0	0	0	8		9=t$
10	1	0	0	10		10=$

Now S_0 is put at the second position of the first bucket: $T_6 = 0$ and *sufinv*[0] = 0, hence increment *count*[0], set *sufinv*[0] = 0 + *count*[0] − 1 = 1, and *B2h*[1] = 1.

i	*Bh* [i]	*B2h* [i]	*count* [i]	*sufinv* [i]	*new_st*[i]	*suftab* [i]
0	1	1	3	**1**	1	0=acbaacatat$
1	0	1	0	**7**	0	3=aacatat$
2	0	1	0	**5**	2	4=acatat$
3	0	0	0	**0**		6=atat$
4	0	0	0	**2**		8=at$
5	1	1	1	**6**	5	2=baacatat$
6	1	1	2	0	7	1=cbaacatat$
* 7	0	1	0	**8**	6	5=catat$
8	1	1	1	0	8	7=tat$
9	0	0	0	8		9=t$
10	1	0	0	10		10=$

Now S_4 is put at the third position of the first bucket: $T_7 = 4$ and *sufinv*[4] = 0, hence increment *count*[0], set *sufinv*[4] = 0 + *count*[0] − 1 = 2, and *B2h*[2] = 1. The bucket is finished. We scan it again to update *B2h*. *B2h*[2] is set to 0, since two suffixes with second character c were moved, but *B2h* should only mark the beginning of the 2h-bucket.

The construction shown can clearly be done in time $O(n \log n)$ and $O(n)$ space. In the original paper Myers describes a small modification of the construction phase which leads to an $O(n)$ expected time algorithm at the expense of another n bytes.

The idea is to store for all suffixes S_i their prefixes of length $T = \lfloor \log_{|\Sigma|} n \rfloor$ as T−digit radix-$|\Sigma|$ numbers.

Then instead of performing the radix sort on the first symbol of the suffixes, we perform it on this array, which can be done in time $O(n)$ since our choice of T guarantees that all integers are less than n. Hence the base case of the sort has been extended from 1 to T.

It can be shown that in the expected case there is only a constant number of additional rounds that need to be performed.

7.12 Appendix: Computing the *lcp* values along with the MM algorithm

This can be done during the construction of the suffix array, without additional overhead, or alternatively in linear time with a scan over the suffix array.

In Myers' algorithm the computation of the *lcp* values can be done during the construction of the suffix array without additional time overhead and with an additional $n + 1$ integers. The key idea is the following. Assume that after stage h of the sort we know the *lcp*s between suffixes in adjacent buckets (after the first stage, the *lcp*s between suffixes in adjacent buckets are 0).

At stage $2h$ the buckets are partitioned according to $2h$ symbols. Thus, the *lcp*s between suffixes in newly adjacent buckets must be at least h and at most $2h - 1$. Furthermore if S_p and S_q are in the same h-bucket, but in distinct $2h$-buckets, then

$$lcp(S_p, S_q) = h + lcp(S_{p+h}, S_{q+h}) \text{ and } lcp(S_{p+h}, S_{q+h}) < h.$$

The problem is that we only have the *lcp*s between suffixes in adjacent buckets, and S_{p+h} and S_{q+h} may not be in adjacent buckets. However, if $S_{suftab[i]}$ and $S_{suftab[j]}$ with $i < j$ have an *lcp* less than h and *suftab* is in $\leq_h$ order, then their *lcp* is the minimum of the *lcp*s of every adjacent pair of suffixes between *suftab*$[i]$ and *suftab*$[j]$. That is

$$lcp(S_{suftab[i]}, S_{suftab[j]}) = \min_{k \in [i, j-1]} \{ lcp(S_{suftab[k]}, S_{suftab[k+1]} \}$$

Using the above formula to compute the *lcp* values directly would require too much time. And maintaining the *lcp* for every pair of suffixes would require too much space.

By using a balanced tree that records the minimum pairwise *lcp*s over a collection of intervals of the suffix array, we can determine the *lcp* between any two suffixes in $O(\log n)$ time (which is sufficient for Myer's online construction).

Since there are only n internal leaves in the tree, for which the *lcp* has to be computed, we spend a total of $O(n \log n)$ time to precompute the *lcp* values.